10-1-1996

A Buffering strategy for stabilizing network data rates

Brian Bell

A BUFFERING STRATEGY FOR STABILIZING NETWORK DATA RATES

by

Brian D. Bell

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:_____
Dr. Charles K. Shank, Assistant Professor

_____
Dr. Tony H. Chang, Professor

_____
Dr. Roy S. Czernikowski, Professor and Department Head

Department of Computer Engineering
College of Engineering
Rochester Institute of Technology
Rochester, New York

October, 1996

THESIS RELEASE PERMISSION FORM

ROCHESTER INSTITUTE OF TECHNOLOGY
COLLEGE OF ENGINEERING

Title: A Buffering Strategy for Stabilizing Network Data Rates

I, Brian D. Bell, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Signature: _____

Date: 11/12/1996

Trademarks

HP	HP is a trademark of the Hewlett-Packard Company
Java	Java is a trademark of Sun Microsystems, Inc.
Netscape	Netscape is a trademark of Netscape Communications Corporation
OSF	OSF is a trademark of the Open Software Foundation, Inc.
Sun	Sun is a trademark of Sun Microsystems, Inc.
UNIX	UNIX is a trademark of X/Open Company Limited.
Windows 95	Windows 95 is a trademark of Microsoft Corporation

Acknowledgments

The author wishes to thank the people that made this thesis possible. First the graduate committee, Dr. Roy Czernikowski, Dr. Tony Chang, and especially Dr. Kevin Shank who guided me through this long process. Thanks also to Paul Ferno who listened to and critiqued numerous ideas and even had a few suggestions of his own. Thanks also to Frank Casilio who was willing to help with systems problems around the clock. Thanks go to Fr. Joe Catanise and Sr. Marlene Vigna for their support and prayers. The greatest thanks go to my parents, Dave and Kathy Bell. Without their support, prayers, and understanding this thesis could not have been completed. This thesis is dedicated to them with my love and gratitude for their sacrifices over the past five years.

Abstract

In the past ten years, there has been substantial growth in the area of networked applications. The expansion of such Internet applications as USENET and the world wide web has lead to greater demand for faster and higher quality data transmissions. However, the Internet tends to have a bursty traffic pattern. Such a pattern interferes with an application's ability to receive a data stream at a constant rate. A constant rate is necessary for real-time, networked multimedia applications to be able to provide a high quality of service to the user. This thesis proposes a buffering strategy that stabilizes a bursty data stream. It uses buffered data to present a client application with a constant data stream. An implementation of the strategy was produced using the Java™ programming language. Results indicate that networked multimedia applications that use this strategy can provide a higher quality of service than applications that do not.

Table of Contents

List Of Figures

List Of Tables

Glossary

Abstract Class — A **class** that cannot be instantiated. Used to combine common methods into one superclass while allowing their implementations to be deferred to subclasses.

API — Application Programming Interface. "An API consists of the functions and variables that programmers are allowed to use to use in their applications" (Flanagan, 1996). Eight packages of classes are provided with the **Java** programming language.

Bandwidth — The maximum rate at which a network can transmit data. It is usually given in bits per second (bps) (Hennessy & Patterson, 1996).

Class — A class is a data structure that is used to specify an **object**.

Client — A client is a process that makes a request of a server for the use of some resource. Depending on the distribution of resources, a computer acting as a client in the case of one resource could be a server for another.

Client-Server — A request/reply model for interprocess communications where system resources are available directly to a limited number of processes on the network, called **servers**. Any other process that needs the resource can act as a **client** and request the use of the resource from the **server**. The **server** will then use the resource as requested and return the results as necessary. The **client** and **server** processes may or may not be running on the same computer.

Extend — Terminology used for **inheritance** in **Java**.

Inheritance — When one class inherits from another, all non-private in the superclass methods and variables become a part of the subclass. The subclass also becomes whatever the types the superclass was. For instance, if Square inherits from Shape, Square gets all of Shape's methods and variables. Square also becomes a Shape. Thus a Square is a type of a Shape.

Internet	A term used to describe the interconnection of the computer networks of most universities, companies, and organizations. The Internet allows users on remote sites to communicate via email, **USENET**, and the **world wide web**.
Java™	"[Java is] a simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, and dynamic [programming] language" (Flanagan, 1996).
MPEG	Motion Picture Experts Group. A technical committee that writes for standards for video compression. Also used to refer to video files compressed according to the standard.
Netscape™	A **web browser.** Currently the most popular browser on the market.
Object	An object is a discrete entity that encapsulates data. Objects are the basis for object oriented programming. An object is a specific instance of a **class**.
Object Oriented	A programming model that focuses on using **objects** to design.
Packet	A collection of data packed together to be sent over a network. A packet often has headers and error correcting information packet with the data. At the level of the Java programs in this thesis, a packet is an array of 1024 bytes.
QoS	Quality of Service. A measure of the performance of a multimedia application.
Socket	An entity that is used for network communications. Sockets provide an abstract view of the network to higher level programs.
Server	In the **client-server** model, the server is the process that has access to some system resource. Resources could be printers, disk drives, etc. The server will accept requests for the use of

	the resource from a **client** and return the results. Results could be a file in the case of a disk resource or an acknowledgment for a print request.
Subclass	A subclass is a **class** that has inherited from some **superclass**.
Superclass	The class the some **subclass** has inherited from.
Throw	In Java all exceptions and errors are thrown to the method that called the excepting method. They continue to be thrown until a method that handles the exception is found. Throw is a key word in Java and is used to manually throw an exception. Errors are not usually not handled in the program. Instead, an error is considered non-recoverable and the program exits.
Throughput	The amount of a network's total **bandwidth** that is available to an application (Hennessy & Patterson, 1996).
trn	threaded read news. A news reader for **USENET** on UNIX machines.
USENET	A world network that provides network news. In USENET news is in the form of articles posted by users into newsgroups.
VOD	Video on Demand. A plan to allow customers to request any video they desire for play at any time they desire.
Web Browser	A program that is used to download data from the **world wide web**.
World Wide Web	The world wide web is a network of interconnected servers that accept data requests from public computers. Data is retrieve using a **web browser**. Abbreviated as WWW or web.

1.0 Introduction

The past few years have seen a tremendous increase in the demand for networked applications. Advances in networks and multimedia techniques have provided designers with the ability to create new classes of applications that were unheard of ten years ago. However, there are still problems left to be solved.

1.1 Multimedia

Over the past decade, as computing power has increased, multimedia has become a focus of the computing industry. Multimedia has been defined as a computer using video, audio, and graphics along with numerical and text information (Minoli & Keinath, 1994). A computer application designer is able to use video, audio, and graphics in the products he designs. This allows programs to take greater advantage of the human ability to use multiple senses at once to assimilate information. Today, online multimedia encyclopedias, with video clips of historic events and with sound clips of famous speeches, are widely available. As computer processing speeds increase and as data storage devices become faster, such applications will become less difficult to implement.

One measure of the performance Multimedia application is the Quality of Service (QoS) of the presentation. Quality of Service is a loosely defined term that measures exactly what it says, the quality of the multimedia playback. Quality of

Service does not have a specific formula. Instead, parameters such as throughput, loss rates, and delay are measured to determine the quality of service (Bowman, et. al., 1994). The higher the quality of service the better the multimedia presentation is going to appear to the user. On local applications running in local memory and off a local disk, current processors provide a quality of service high enough for video and audio to be used.

1.2 Networked Multimedia

Along with the increase in multimedia research and multimedia products in the 1990s, there has also been an increase in networked applications. A networked information is one that retrieves some type of information from over a network. The information is then used by the requesting application. Such applications typically run as client-server applications. One computer is a server and has the required information stored on its local system. Any other computer on the network can act as client and request that information from the server. Long before multimedia applications became popular, single media text based applications followed this model. For example, USENET readers, such as trn (threaded read news, a news reader for UNIX based machines), retrieve articles from a local news server and display them and display them for the user.

The advent of multimedia techniques has added to the types of networked applications that exist. Instead of retrieving simple text files from servers, clients are

now requesting video and audio files. In the case of local networks this change does not present a large problem.

The growth of the Internet, however, has lead to large numbers of applications that retrieve data from servers that are not located on the same local net as the client. Browsers for the World Wide Web, such as Netscape™ allow users to download video, audio, and images from sites around the world. The ability to retrieve non-local multimedia presents designers with enormous potential for the transmission of information. There have been proposals for systems that would allow for full length movies to be accessed over a network by consumers in their homes. These proposals should not be confused with currently existing pay per view systems. The new system, called video on demand (VOD), would allow customers to choose to watch any movie that is on the server at any time. VOD systems are still in the theory stages and will require further research to make them feasible.

Another type of networked multimedia application that is receiving large amounts of attention is video conferencing. Such systems allow people at two or more remote sites to confer using video and audio over a network. Such systems do exist today but they tend to be expensive and require large private networks (Minioli & Keinath, 1994).

One of the primary problems that limits networked multimedia applications is the need to maintain an acceptable Quality of Service. Currently existing Internet

multimedia applications will generally download entire files from servers before playing them for the user. With the entire video or audio file available in local memory or on a local disk, the application will be able to supply a sufficiently fast and stable data rate to the media player.

While this method works well for short clips, it is not practical for lengthy sound and video files. A three minute sound file may be two megabytes in size. A video file of the same length would be only 30 to 60 seconds. Any two megabyte file will take minutes to download. To display a full length motion picture in a VOD system, the client would need to wait for hours and have an unreasonably large amount of local data storage capability available. Therefore for sizable multimedia files, the most reasonable method is to play the file in real-time, that is, to decompress and play the data as the packets are downloaded. This eliminates both the long waits and the need for large storage devices at the client. In the case of video conferencing, playback must be in real-time. It does not make sense to have a video conference where the participants cannot view the video sent from the other conference sites until the conference ends.

The term real-time refers to an action being observed in the real (same) time that it happens. For any multimedia application there are two things that can happen in real-time. The first is the decompression of the data file. The decompressing of the file happens as playback occurs. As the earlier parts of the file are being played, the

following sections are being decompressed into the format used by the actual playing device (the speaker or the monitor). This differs from a system where the entire file is decompressed before any data are sent to the device. Today, most applications use real-time decompression for local files.

The second method of real-time operation is only applicable to networked applications. In this case, real-time playback can be done while the download is in progress. As the packets arrive they can be decompressed and played. This type of real-time transmission is the type necessary for video on demand and video conferencing systems.

The problem with real-time playback of arriving multimedia is the network bandwidth requirements. Video requires 1.54 million bits per second (Mbps) for successful playback. Audio requires a less demanding, but still difficult to maintain, 100 - 300 Kbps. The web browsers run at RIT usually achieve 300 1000 Kbps. The Internet can deliver a higher data rate, but only if the entire bandwidth is available for one transmission. The problem develops when the network's entire bandwidth is not available due to multiple transmissions.

As the number of users requesting transmissions across a network increases, the amount of time that the network can spend servicing each transmission decreases. If a network can sustain a stable bit rate of 50bps, then if there is only one transmission on the network it can proceed at 50bps. If a second request for use of the network

occurs, then there may be a loss of data rate from the first transmission. The total bandwidth of the network remains the same but it must be shared by multiple transmissions. As more and more transmissions are requested across the network, the bandwidth available to the initial transmission, as well as to any other, will decrease. What will result is a network traffic pattern that is bursty. A client will receive bursts of data from the network followed by a period of no data while other users' traffic is carried over the network. This bursty traffic can destroy any quality of service in a real-time playback. The user would hear a few seconds of sound followed by a period of silence until the next burst is received. The result is a choppy playback that most users would find unacceptable.

1.3 The Buffering Strategy

This thesis presents a buffering strategy designed to counter the effects of bursty network traffic. Using buffers both at the server and at the client increases the media player's tolerance for bursty data. First, a multimedia program requests a file be sent at some data rate. The server buffer initial begins to buffer some bytes and then sends them over the network to the client buffer. The buffer at the client side initially buffers portion of the file before playback begins. Then it begins to send the requesting multimedia program packets at the requested constant rate, refilling its buffer as more data bursts arrive. The client buffer also sends feedback to the server buffer informing it of its current state. Using that information, the server buffer can choose to change the

rate it is sending data to the client. This gives the server buffer the ability to prevent an overflow of the client buffer. This strategy requires that the user be willing to wait for a short time before playback. During that waiting time, the buffers are filled with enough data to ensure that the required data rate can be maintained during the periods when the available network bandwidth is low. This initial buffering wait does result in a slight decrease in the real-time nature of the playback. However, for most applications a short lag is an acceptable tradeoff for a constant data rate.

To test the theory of the strategy, an implementation is required. For this thesis, an implementation was done in the Java™ programming language and tested on a network of HP™ workstations. Results indicate that the strategy allows for successful playback of audio files when the available bandwidth drops well below that which will be successful without the buffers.

1.4 The Java™ language and Object Oriented Design

The Java language was chosen for implementation of the buffering strategy. This choice was made for three reasons. First, Java is an object oriented language. Object oriented languages are based on the object model of programming. In that model all data structures, methods, variables, functions, etc. are encapsulated into objects. The objects are then used as the primary building blocks of the program. When used on the proper type of problem, object oriented design techniques can make

the design and implementation of the program easier than if a structural language is used.

The second reason for choosing Java is its high level Application Programmers Interface (API). The API provides high level classes that greatly simplify communication and graphical user interface design. These high level abstractions tend to make program design and implementation easier.

The existence of a socket API is the third reason for choosing Java. Java was designed to be a networked language. It seemed logical then, that an implementation of a network related strategy would be done using a network based language.

As the implementation progressed, it became apparent that there were some disadvantages in using Java. The same API classes that simplify the use of sockets and streams also encapsulate all the details of their operation. One of the hidden details is an attempt by Java to make network communications smooth by doing its own buffering, Occasionally this interfered with the necessary operation of the program. In that case it would have helped to have a lower level network interface.

Another disadvantage of Java was bugs. Java is a very new language and there are still bugs in the language itself. Sometimes a bug in the current version of the language required a change in the design to compensate. This lead to some limits in the functionality of the Java implementation. None of the limits, however, if very significant and all should no longer be a problem as Java improves.

The other possible choice, based on knowledge of the language and compiler availability was C++. C++ was not chosen primarily because it does not have a set of standard classes comparable to the API.

1.4.1 Booch Class Diagrams

Booch class diagrams are used to show the relationships between classes in a system. They are detailed by Grady Booch in Object Oriented Analysis and Design (Booch, 1994). The diagrams allow designers to show how the various classes in a system interact with and relate to each other. Each class is represented by a cloud with its name and major attributes inside. Classes are connected using various types of lines. Inheritance is indicated by an arrow. One class containing another is shown by a line with a solid circle. A line with a hollow circle denotes that one class is using another. Figure 1 provides an example Booch diagram.

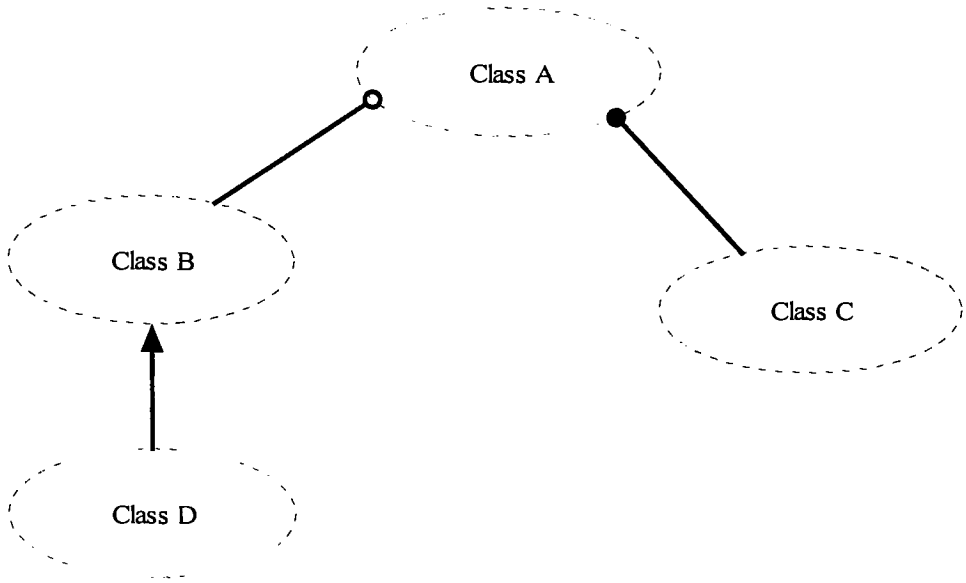

Figure 1 - Sample Booch Class Diagram

In this example class A contains an instance of class C. This is signified by the solid circle at the end of the line by Class A. Class A also uses an instance of class B, which is signified by the hollow circle. Class D inherits from class B. This is signified by the arrow pointing at Class B. It is possible that the actual object used by class A is of type class D, but it sees that object as a B.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 presents some works related to this thesis. Chapter 3 details the buffering strategy greater in detail. Chapter 4 discusses the design and implementation of the Java version of the strategy. Chapter 5 discusses the testing of the implementation and an analysis of its success. Chapter 6 concludes this thesis and makes some remarks about possible future works.

2.0 Related Work

There are a number of projects and products that are related to this thesis. The strategy presented in this thesis is one way to deal with some of the issues involved in developing a networked, real-time multimedia application. Others have suggested different ways to approach the problem. Some research systems have been published. In addition, there are some real-time, networked multimedia products available commercially.

2.1 Real Audio™

Real Audio is perhaps the most successful commercial, real-time, Internet sound player available. The makers, Progressive Networks, refer to Real Audio as an Audio on Demand (AOD) program. It is the same theory as proposed VOD systems. Any user can request any file for playback at any time. Progressive Networks claim that Real Audio can provide broadcast quality sound over the Internet and near CD quality over a ISDN line. Personal experimentation shows that version 2.0 does provide sound on demand in real-tine; however, the quality of the sound is highly variable. At times the player will provide a high quality AM broadcast-like transmission. When the bandwidth available to the audio stream drops or when packets are lost, the quality quickly slips to the point where the audio is unintelligible.

Real Audio attempts to counter the problems involved in real-time network transmissions in two ways. First, the player requires that all sound files be in a specific Real Audio format. This format is a lossy compression of standard .wav or .au audio files (Lawrence, 1996). The compression reduces the amount of data that has to be sent over the network. With less data to be sent over the same length of time less bandwidth is required for the transmission. The second approach Real Audio takes is to have the player buffer data packets for ten seconds before beginning to play the audio file. This appears to be similar to the strategy presented in this thesis. It is, however, difficult to be sure exactly how close the parallel is because specific algorithms of Real Audio have not been disclosed.

Progressive Networks provides their client audio player free of charge. This policy allows anyone with an Internet connection to download and play Real Audio files. If one wishes to develop a web site that containing Real Audio files, a server must be bought from the company. The server has the capability to send a Real Audio data stream over the Internet for a real-time playback. Without the Real Audio server, playback occurs locally -- after the entire file has been downloaded.

Despite the cost of starting a Real Audio web server, the format has become very popular. National radio networks, including ABC and National Public Radio, have broadcast real-time programming over the Internet. ESPNET SportsZone broadcasts sports talk shows and occasional play by play using Real Audio. There are

thousands of other web pages offering Real Audio sound. Progressive Networks claims over 10 million players have been distributed. The recently released version 3.0 has been designed to substantially increase the quality of the audio service. It seems quite likely that this format of Internet audio on demand will continue to grow and to improve.

Although there are similarities, Real Audio is not entirely compatible with the strategy presented in this thesis. One of the goals of the strategy is to provide a solution that works independent of the application and the data type. In contrast, Real Audio only works for audio files encoded in the Real Audio format. The client buffer in Real Audio is a part of the player where it is independent in the proposed strategy. Whether there is any type of server buffer is unknown. Still, while Real Audio does not entirely follow the strategy, the parallels suggest that this thesis' strategy is viable.

2.2 University of Oregon MPEG Player

A group of researchers at the Oregon Graduate Institute of Science and Technology has developed a distributed real-time MPEG video player (Cen, Pu, et. al., 1995). The program consists of a server and a client. Each process is located on a separate machine. The server sends the video to the client through a network. The client contains a buffer, a controller, a MPEG decoder, and a video player. It receives the data stream from the network, decodes the MPEG frames and displays the video on the screen. They also built in the ability to receive audio synchronized with the video.

They use a buffer at the client end to deal with data rates that are greater than the decoder can handle.

Although the final result of the project was a distributed real-time MPEG player, the primary focus was software feedback in networked applications. They developed a software toolkit for use with software feedback. The toolkit consists of lowpass, highpass, and other filters; comparators; and other modules. The modules are used to process the feedback about data input that is sent from the client to the server. The result is a steady stream of information concerning the data reception patterns of the client. The toolkit was then tested by using it to develop the MPEG player programs. The software feedback is used to determine how well the client is handling the data stream from the server. If the data stream is overwhelming the player, the server knows to scale back the send rate. If there is too little data being received, the server increases the output. The buffer stores excess data when the rate is high and supplements the input when the rate is low until the server can respond to the feedback.

The quality of service measurement they used to test the functionality of the MPEG player is smoothness. It is based on frame rate and on the number (and type) of frames dropped (received but not played). Playback is defined as perfect when smoothness is equal to zero. With their software feedback active they achieved a

smoothness of 5 in a congested network. Without the feedback the smoothness was 25 under the same conditions.

At first, the video player seems very similar to the strategy proposed in this thesis. There are, however, some differences. Like Real Audio, this player is a specific program for a specific type of media. The buffers and feedback are built into the server and the player. The strategy proposed here does not suggest this. Instead it proposes that the buffers run as separate programs. This allows existing applications to take advantages of the stable data rate offered without requiring revision of the application.

The MPEG player also places a greater emphasis on the feedback aspects of the system. The buffering strategy does use feedback but it is only one part of the system. Instead, a greater emphasis on the client buffer is made in the proposed strategy.

3.0 The Buffering Strategy

3.1 Computer Networks

Network communications are controlled by protocols. There are protocols that define how packets should be formed, that specify what information should be in packet headers, and that define how to specify the network address of a computer. These protocols are usually formed into layers. For example, at the lowest layer there is a protocol defining how to place individual bits onto the physical medium. At a higher layer, the bits are formed into packets. That layer makes sure that the packets are received in order. When packets do arrive out of order, the layer does whatever is necessary to reorder them. At a higher level, there are protocols to determine the best route for data packets to be sent when traveling from the source to the destination. Another layer presents -- to a higher layer -- an abstraction used to open a connection to another computer and to transfer data.

The buffering strategy proposed in this thesis view the network from a high layer. This approach allows the strategy to concentrate on buffering not on managing a network. This resulting system is shown in Figure 2.

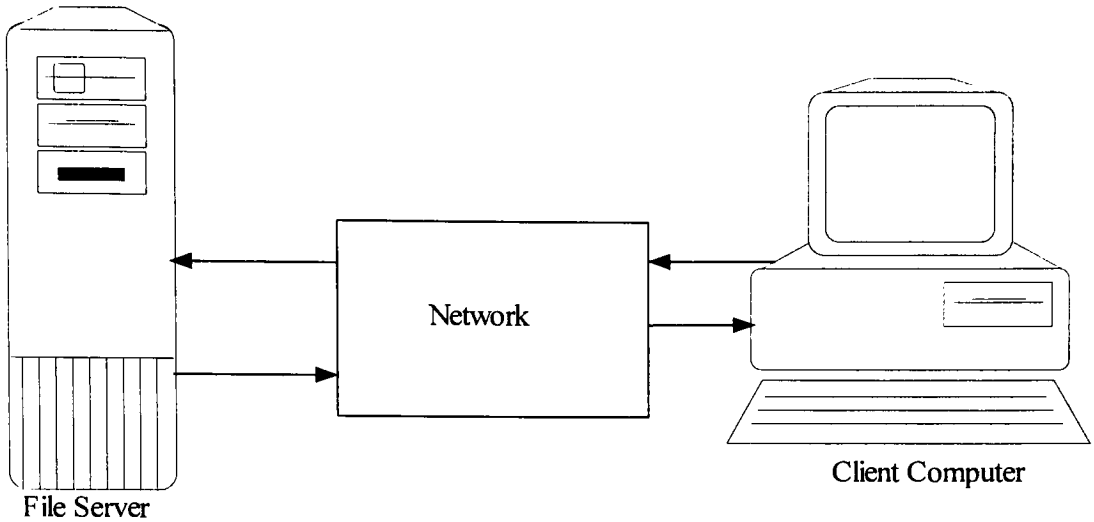

Figure 2 - The Strategy's View of the Network

To the strategy, the network is a black box. No details about the network are known. Opening a network connection requires no more than asking for a connection to an address. All connections appear to be errorless. Packets should not arrive out of order nor should any packets be lost and fail to arrive.

At a lower level such assumptions would not work. They can be made, however, if the black box network corrects all errors passing the data to client. To do so is possible. It is not, however, without consequences.

Fixing errors takes time. If there is a need to wait while a packet is being resent, then the client will see a network that has stalled.

3.2 Bursty Network Traffic Patterns

If the data stream stalls while it waits for the network to fix an error, the result may be a bursty network traffic pattern. A bursty pattern is observed when the rate at

which data is received is not constant. Data is received for a short amount of time followed by a period where little or no data is received. The resultant pattern can be seen in Figure 2.

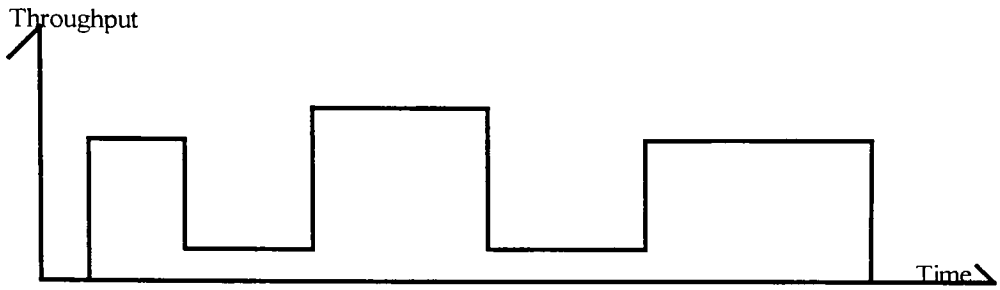

Figure 3 - Sample Bursty Traffic Pattern

Throughput is the amount of the network's total bandwidth that is available to an application (Hennessy & Patterson, 1996). The periods where the throughput is high are the bursts. The length of both the high and low throughput periods varies depending on the network's conditions. During the bursts, the throughput may be equal to the entire bandwidth. It will be high enough to supply the application with the required data rate. Similarly, the stall times will not necessarily result in a zero data rate. The data rate will be low enough; however, that whatever application is receiving the data will not be sufficiently supplied.

There are multiple causes of bursty traffic. The time spent correcting transmission errors was mentioned as a cause before. A second is the amount of the network's total bandwidth available for the transmission. As the amount of traffic on a

network increases, less bandwidth is available for each individual transmission. A second consequence of more traffic on the network is an increase in transmission errors. More traffic increases congestion, which will lead to more lost packets and more packets arriving out of order. The increase in errors will lead to more waiting for data. Those waits will increase the bursty nature of the incoming data.

In chapter 1, the effect of bursty data rates on applications was discussed. If data arrives in a bursty pattern there likely will be times when there is no data available for the client. In situations where the data is used in real-time, periods where no data is available can cause severe performance degradation. At the same time, it is possible that during the bursts more data will arrive than the client can handle.

3.3 Buffers

3.3.1 The Client Buffer

If real-time applications are going to be effective, there must be a method of compensating for bursty network data. One possible approach is to buffer the incoming data. Most programs that receive data off a network use buffers. Buffering the incoming data prevents data loss when packets arrive faster than they can be consumed by the program. During periods of high throughput any excess data is buffered. That data can be processed during the low throughput period that eventually follows. This approach provides some smoothing of the average data rate but does not

solve the problem. If the burst is too short or the stall period is too long, the buffer will eventually underflow and no data will be available for the application to process.

Aside from the danger of underflow, using a buffer appears to be a good solution for bursty traffic. If no buffer is used the server can certainly send data at the rate the client requires. There is no guarantee, however, that the client will receive data at that same constant rate. Indeed, a bursty traffic pattern makes such an event unlikely. With a buffer, even if the network disrupts the data rate, the buffered data can be used to supply the client at the desired rate. Therefore, when the buffer is not empty the application can proceed regardless of the current arrival rate. The bursts will not be noticed and the application can be supplied at its optimal rate.

The solution to the buffer underflow problem has not yet been mentioned. It can be avoided by initially buffering some data. Once the first packet arrives from the network, the requesting application could begin to process it. If it waits for a while instead, a substantial amount of data will build up in the buffer. Once the processing starts, the buffered data can be used to supply the difference whenever the arriving rate drops below the rate required by the application. When a burst raises the arrival rate above the required rate, the excess received can be used to replace the data that were used during the low throughput period.

3.3.2 The Server Buffer

It would be helpful if the condition of the network were determined by the server at the start of the transmission. The server would then be able to use that information to determine at what rate to send data in order to get an acceptable average data rate at the client's end. If the average rate is at least equal to the rate the application is using, then the client buffer can stabilize the data rate seen by the application. However, there is no guarantee that the network conditions will remain constant over time. The network could become less bursty, resulting in a higher average rate. It could also degrade, causing a lower average rate to be received. If the server continues to send at the same rate, the client buffer will either underflow or overflow. Neither condition is desirable.

The actual server program, that is the program that reads the file and places it on the network, does not vary its send rate. In order for the server side to adapt to changing network conditions, this functionality must be added. This strategy proposes that a second buffer be added, this time at the server's end of the network. The server program can send packets to the buffer at its fastest possible rate. Those packets will be buffered in the server buffer. Initially, the server buffer starts sending data at the client's requested rate. As long as that send rate provides a sufficient average rate to the client, no changes are required. When network conditions change, the server buffer can modify its send rate to compensate for the changes. If the client buffer is emptying,

the server buffer can use some of the buffered data to increase the send rate. If the client buffer begins to fill up, the server buffer can scale back its send rate to avoid overflowing the client buffer. The server should have more resources available to devote to buffering than the client. This means that the server buffer can be larger than the client buffer. Even if that is true, care must be taken to assure that the server buffer does not overflow.

3.3.3 Feedback

The two buffer strategy proposed thus far requires that the server buffer have knowledge of the client buffer's condition. That information is not available to the server under normal circumstances. After the file is requested, the only thing a server receives from a client is a low level packet acknowledgment. This strategy proposes that feedback be used to send the required information from the client to the server. The client buffer should send the server buffer feedback containing updates on the current status of the client buffer. The server buffer can use those updates to determine if the send rate needs to be changed.

One question to consider is what information should be included in the status messages. There are several possibilities. Whatever information is chosen should provide the server buffer with as much information on the status of the buffer as is possible while sending the least possible amount of data. The client buffer could send the current number of packets buffered. That would let server buffer determine

whether the buffer is in danger of underflowing. It would not prevent overflow unless the server buffer knows the size of the client buffer. The average delay between packet reception is another possibility for the feedback. The average delay would be enough for the server buffer to determine the client buffers receive rate. It could use that information to determine if the rate is fast enough. However it would not give any information on the amount of packets buffered. The elapsed time since the last packet arrived is another option. The problem with this is twofold. First, there is no information on the condition of the client buffer. Second, only sending the most recent gives the server buffer no history. It cannot determine if a long delay is indicative of a degradation of the network or just a one time glitch. It could also send an update every time a packet is received. This forces the server buffer to calculate the rate and does not solve any of the previous problems.

This strategy suggests that the client buffer needs to send two statistics, the average rate at which it is currently receiving data and the percentage of the buffer that is filled. No further information is needed. The server buffer needs to increase the send rate if the current receive rate is too low. If it is too high, the send rate should be decreased. The percentage of the client buffer that is currently full lets the server determine how much to change its send rate. If the client is receiving at too low a rate but the client buffer is 60 percent full, most likely the server buffer does not need to increase the send rate. If the client rate is too low and the client buffer is only two

percent full, however, the server will need to increase the send rate immediately to avoid underflow.

3.4 Relationship of the Buffers to the Network

Thus far no comment has been made concerning how the client buffer and the server buffer should interact with the network, with the client application, and with the file server program. It is possible to make the buffers a part of the client and server programs. If a designer is developing a completely new application, including both new server and client programs, then such an approach would work. Under that plan preexisting applications could not use the buffers. A method that applies the buffering strategy to those applications as well is desirable.

To do so is not difficult. Each buffer has to be placed in its own individual program. Both programs must also include implementations of the controlling algorithms that are required to send and to receive data; to send or receive feedback; and to calculate the required send rates. Each program should be run on the appropriate computers as daemon processes. Daemon processes run on a computer at all times. Once started, the daemon sleeps until another process requests that it do its job. When it is done doing the job, the daemon goes back to sleep. This method allows the a program to always be running without using large amounts of system resources when it is not being used.

If a computer might act as a file server, the server buffer daemon should be started on it. If a machine acts as a client, the client buffer should be running on that computer. Figure 4 shows how using the daemons modify the network that was shown previously in Figure 2.

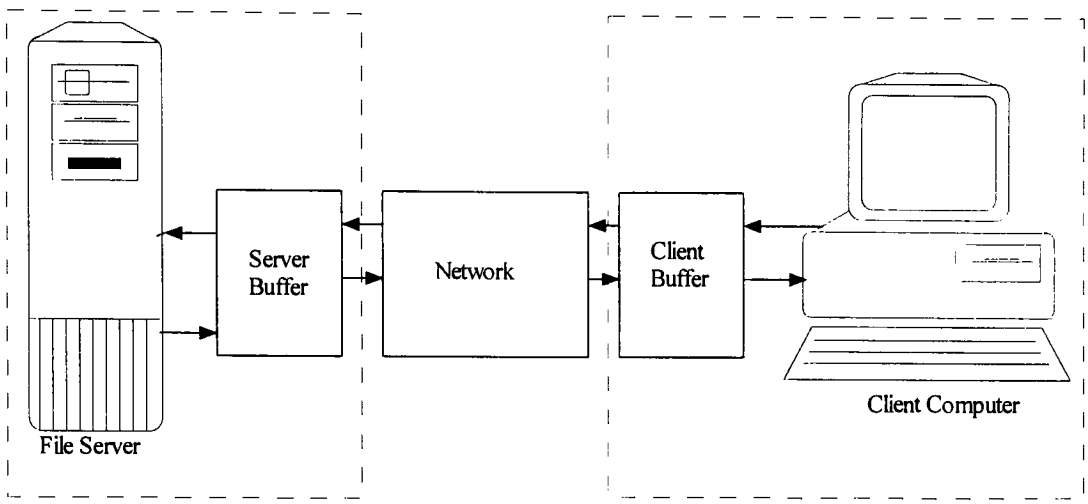

Figure 4 - The Buffer Strategy

This is the system as it actually exists. The file server is not necessary a separate machine from the one that is running the server buffer. Similarly, the client buffer may be running on the client computer. It will definitely be running on a computer that is on the same local network as the client computer. The dashed lines denote the local nature of the buffer with respect to its application or server.

Figure 4 also shows how the network views the strategy. From the network's view, everything inside the dashed lines is one single system. The server buffer is its access point to the server; therefore, it is the server. The same is true for the client. The

client buffer contacts the network and receives the network's response. So, to the network, the client buffer is the client computer.

The server daemon should be designed to accept all connections from the network. It should act as a gateway from the network to the actual server. This means that all server requests will automatically use the buffering strategy.

The client buffer daemon should be designed to mask the network from the application. Any process that contacts the network to get a file will really be contacting the client buffer. This allows the client buffer to work without client applications ever realizing that another process is working to stabilize the data stream. Therefore, from the client application's perspective the system looks like Figure 5.

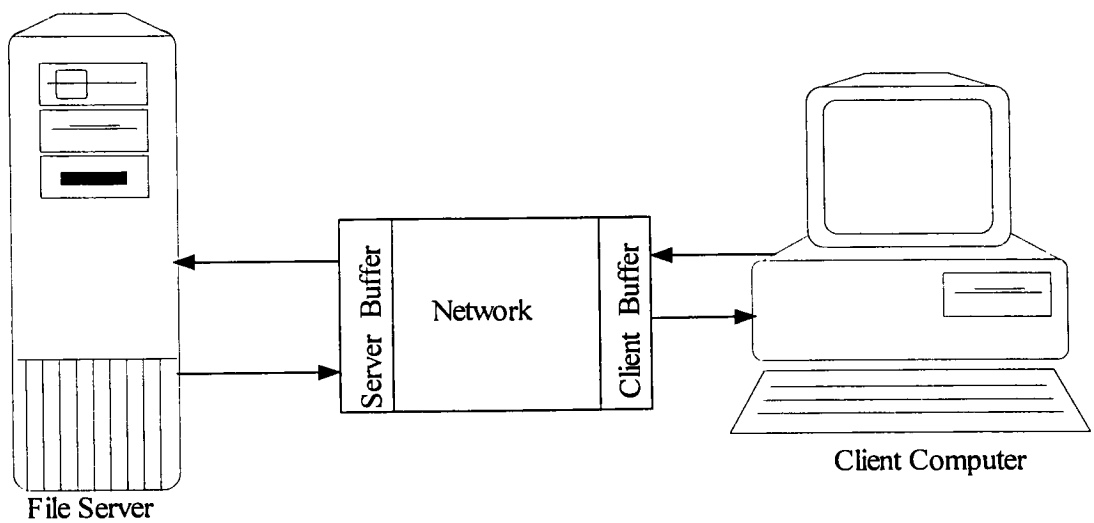

Figure 5 - View of the Network According to the Client and Server

To the client application, the client buffer is indistinguishable from the network. To the server, the server buffer appears to be the part of the network to which it talks.

Some consideration must be made concerning which computers should have daemons running on them. Most local networks have one gateway machine that is the only computer physically connected to the outside world. That computer may or may not be the file server for that network. Regardless of whether it is the file server, one server buffer running on that computer is sufficient. Any non-local file requests will already be going through that computer. Therefore, the server buffer can receive the request there and forward it to the server.

For the client buffer, two approaches are possible. The first would be to run one daemon on the gateway machine. If the bandwidth of the local network is sufficient, using just one process can save resources. If not, an individual client daemon must be run on each local network computer that will be running networked real-time applications.

3.5 Consequences of the Strategy

The primary consequence of using this strategy with real-time, networked applications is the wait required while the client buffer is pre-filled. This wait is going to affect the Quality of Service provided to the user. Accordingly, it must be partially under the control of the user. Increasing the timer spent buffering before playback,

increases the amount of time that the strategy will be will be able to deliver a constant data rate regardless of network conditions. The corresponding improvement of the quality of service during the playback is partially offset by the loss of quality that is inherent in forcing the user to wait for the playback to start. If the user is not willing to wait for any length of time the system will not be able to guarantee a constant stream. This gives the initial impression of a high quality of service. It is, however, an impression that will soon be lost when the lack of pre-buffered data causes the quality of service to crash. The best method would be for the user to give the maximum time that he is willing to wait for the playback to begin. The user should also specify the quality of service level he expects. If the strategy has enough data (based on current conditions and the QoS requested) buffered before the end of the allotted time period, it can start the playback sooner. If, when the maximum allowed wait period expires the buffer does not have enough data to proved the quality requested, the user can either accept a reduced quality of service or terminate the transmission.

A related effect of the delay is to lessen the real-time nature of the application. Once the initial buffering is complete and the application starts, the data will be delivered to the user with a lag equal to the initial delay. When the application is receiving preexisting data files the lag is not a concern. Once the application starts to play, as long as the playback is smooth and the data rate is constant, no one will care if the frame or note being played was actually received a few seconds earlier.

The lag is of greater concern for live applications. If a live broadcast over the net is being received by the client, the user is going to want to hear the broadcaster as soon after he speaks as is possible. People want to hear the ball game called as it happens, not after a delay. Here, again, the user should have the final say. If the user is willing to hear a play by play that lags the action by one second, the strategy will be able to guarantee a certain level of quality for the playback. The level will be determined by how much can be buffered during the initial wait and by the conditions of the network. If the user is willing to wait longer and to have a longer lag, then the strategy may be able to present a higher quality of service during playback. However, if the network is bad enough then it is possible that no amount of buffering will give the user a high QoS. There will come a point where using further resources for buffering a larger amount of data will only delay the inevitable point where the buffer empties and the QoS falls.

Interactive and two-way applications such as video conferencing are the least tolerant of the lag caused by buffering. A conference where both participants are getting their data after a delay could be confusing. For that reason, this strategy may not be appropriate for two way applications.

Another consequence of using the buffering strategy is the overhead the programs require. When the daemons are active, they can use a large amount of system resources. Each daemon has to calculate send rates, receive and buffer data, and send

data at specific intervals. That much activity can use a considerable amount of the processor. Also, if the internal program overhead required to maintain the buffers and to calculate the feedback parameters is sufficiently high, the maximum possible data rate may be lower than that which could be sent without the buffers. Next, the buffers are likely to use a large amount of memory. Since data rates for most applications are in the kilobyte or megabyte per second range, the size of the data that is buffered initially will be substantial. Therefore it may be necessary to limit the size of the buffers to prevent them from using too many resources or even running out of the buffering resource (memory, disk space, etc.) altogether.

4.0 The Java Implementation

4.1 Java Features

The Java language has a number of features that were useful in designing the implementation of this thesis. Most were not originally proposed for Java. Instead, they are ideas that were introduced in other languages and adapted for use in Java.

4.1.1 Packages

A package is a group of classes that are related to each other. Classes that are a part of the same package have greater access to each other's methods and variables than non package member classes have. When developing programs, usually classes are taken from many preexisting packages, including the Java API. In the case of this implementation, the main classes for each program are contained in one package. Other classes that are used are imported from the API or from a utility package developed for the thesis.

4.1.2 Interfaces

Another feature of the Java language is interfaces. Interfaces are similar in function to abstract classes but are not classes at all. They contain methods and variable declarations, but no implementations. To use an interface, a class declares that it implements that interface. Then it must provide an implementation for every method in the interface. A class can implement multiple interfaces. If a class implements an interface the it can be referenced as if its type were the interface's name.

4.1.3 Threads

A thread is the basic unit of control in a program. Each program has at least one thread. That thread executes the instructions provided by the program and exits when all instructions have been completed. In some cases it is desirable to have multiple threads for one program. Multiple threads allow a program to execute many different instruction paths at the same time. There are two ways for threads to be implemented. In the first, they can all be independent with each having its own operating system process. The alternative is for all threads to operate as a part of the same program. The program is one operating system process and all threads share the same address space (Booch, 1994). This is the model that Java uses.

When a multithreaded program is running on one processor, only one thread can be run at any given time. All other threads must be sleeping. A sleeping thread has no access to the processor. It does not execute instructions. Instead, it must wait until it is awakened and given a chance to run.

Because only one thread is being run at any given time, threads on a single processor machine are not truly concurrent. Instead, each thread must be given the processor for a short amount of time. This taking turns gives the illusion of concurrent execution (Booch, 1994).

There are two possible ways to schedule threads. One is called preemptive multithreading. Under this plan, the scheduler forces all threads to give up the

processor after a certain length of time. The second, non-preemptive or cooperative multithreading, does not force a thread to suspend its execution in favor of another. Instead, the threads must willingly relinquish control of the processor either by calling a sleep or yield command or by blocking to read from an input device. For either scheduling algorithm there needs to be a scheduler to determine which thread should be granted the processor when more than one thread is ready to run. Under a preemptive scheduling algorithm, the scheduler must force a thread to give up the processor if it has been using it for too long. Usually the scheduler runs as a system thread that cannot be started or stopped by the user. It only awakens when it needs to schedule a thread or make a preemption.

Java is a multithreaded language. A thread can be created by instantiating the Thread class. When created, the thread object is given an object that implements the API supplied Runnable interface. When the thread is started, the Runnable class' run method is executed by the new thread. The thread continues to execute (when it is not sleeping) until the run method ends or the thread's stop method is called. Java's threads can also be given priorities. When it is time for the scheduler to pick a thread to run, higher priority threads will be chosen over lower priority threads

The Java language specification does not state whether the scheduler should be preemptive. The result is different Java platforms use different models. Windows 95's version, which was created directly by Sun, uses a preemptive thread scheduler. The

HP version – the platform used for this thesis – uses a cooperative scheduler. The HP version is a port done by the Open Software Foundation (OSF). Fortunately, a cooperative scheduler was not difficult to design for because all the runnable objects used tend to need to sleep for certain periods or to block to receive data from a socket.

4.2 Utility Classes

In any medium or large programming project, there are going to be sections of code that are used multiple times. For this project, there were a few classes that were used in most or all of the programs. Instead of making those classes a part of each package they are used in, it was more efficient to place them into one utility class that all packages could use. These classes were the first classes developed for the thesis. This made it possible to use them in the design of the other classes.

4.2.1 IO Class

The IO class is probably the class that is used most often in the project. It encapsulates everything related to network sockets. Network sockets are used to communicate with a network. One socket connects to a second socket. The IO class contains a socket that can be connected to any computer that has a thread listening for a connection. When a specific IO object is created, a host name and a port number are specified. The object opens a connection to a listening socket on the specified host computer. After the connection is established, the object creates an input data stream and an output data stream. The IO object uses the two streams to send and receive data.

The streams are not available for other objects to use. Instead, IO provides fourteen methods – six to send and eight to receive – that allow other objects to transfer data. This approach is consistent with the object programming model, in which objects provide access to private variables through public methods. Figure 6 is Booch class diagram for the IO class.

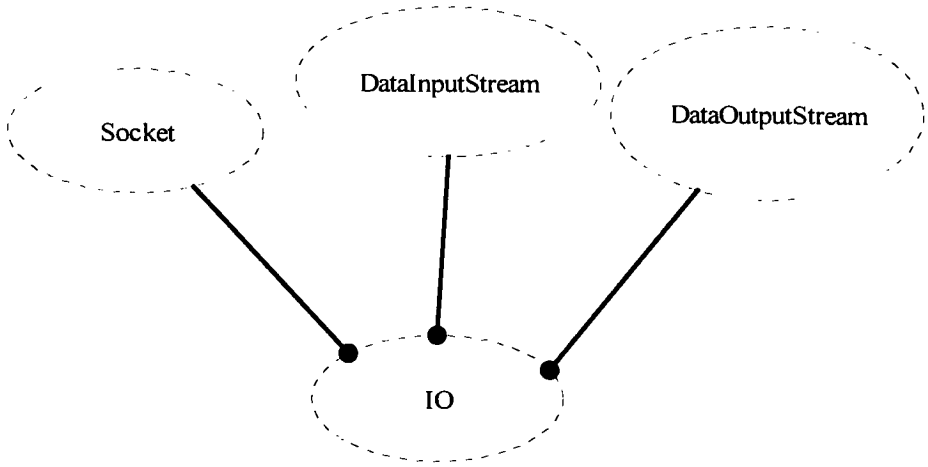

Figure 6 - Booch Diagram of IO Class

4.2.2 Buffer Class

The Java API includes a Vector class. The Vector class is a resizable array of Objects. Its maximum size can increase or decrease as needed. This allows the vector to minimize its memory usage. The Vector class a has large number of methods that can be used to access an object in the vector. Once an element has been accessed, it can be removed from the buffer using a separate remove method.

The Buffer class was created for this project. It extends the Vector class. In addition, it makes changes that add new options to the class. First, Buffer has a strictly enforced size limit that can be specified when the object is constructed. If a maximum size is specified, only that number of objects can be in the buffer at any given time. If no maximum size is specified, the maximum integer value is used as a default. If an attempt is made to add an object to an already full buffer, a BufferFullException will be thrown. Buffer adds a method that allows users to check whether the buffer is full. Buffer also takes two of Vector's instructions and encapsulates them into one instruction. The BufferOut method is used to both get a reference to the first object in the buffer and to remove that object from the buffer. This change gives the class a queue-like object removal structure. It is used with the BufferIn method to give it a First in-First Out (FIFO) structure. Figure 7 is a Booch diagram for the Buffer class.

All of the classes in Figure 7, except for Buffer, are part of the API. Object is the ultimate super class for all Java objects. The A inside the upside down triangle in the Dictionary class denotes an abstract class.

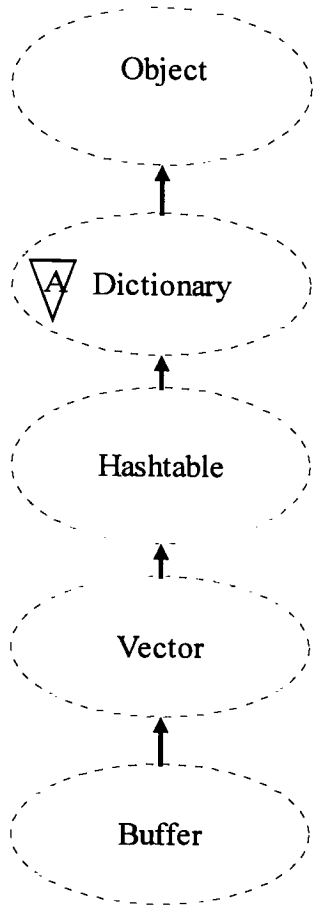

Figure 7 - Booch Diagram of Buffer Class

4.2.3 Statistic Class

The Statistic class provides a small number of methods used to analyze an array of numbers. The first method adds all the elements of an array and returns their sum. This provides an easy way to calculate the sum of any number of integers. Instead of using a FOR loop in the code, a programmer can make a one line call to Sum. The second method also takes an array of numbers. It returns the mean of the numbers in the array. The third method will, given an array of numbers and their

mean, return the standard deviation of the numbers. The class was written for simple statistical analysis of the elapsed time between packet receptions. It is, however, used in other classes. It is described here because it is a part of the utility package.

4.3 The Core Classes

The core classes are those classes that make up the primary core for both the client and the server buffers. Both buffer programs contain the entire core. The classes interact with each other using the same connections for both programs. However, for some of them, the exact implementations of the client and the server versions are slightly different. This variation is necessary because different helper classes are used in the two programs. The differences require that two versions of each class exist, one in the server buffer package and one in the client buffer package. Having two versions is not the best way of designing the packages and was not initially intended. It proved necessary, however, due to changes in the design. The Booch diagram is the same for both versions of the core and is included as Figure 8.

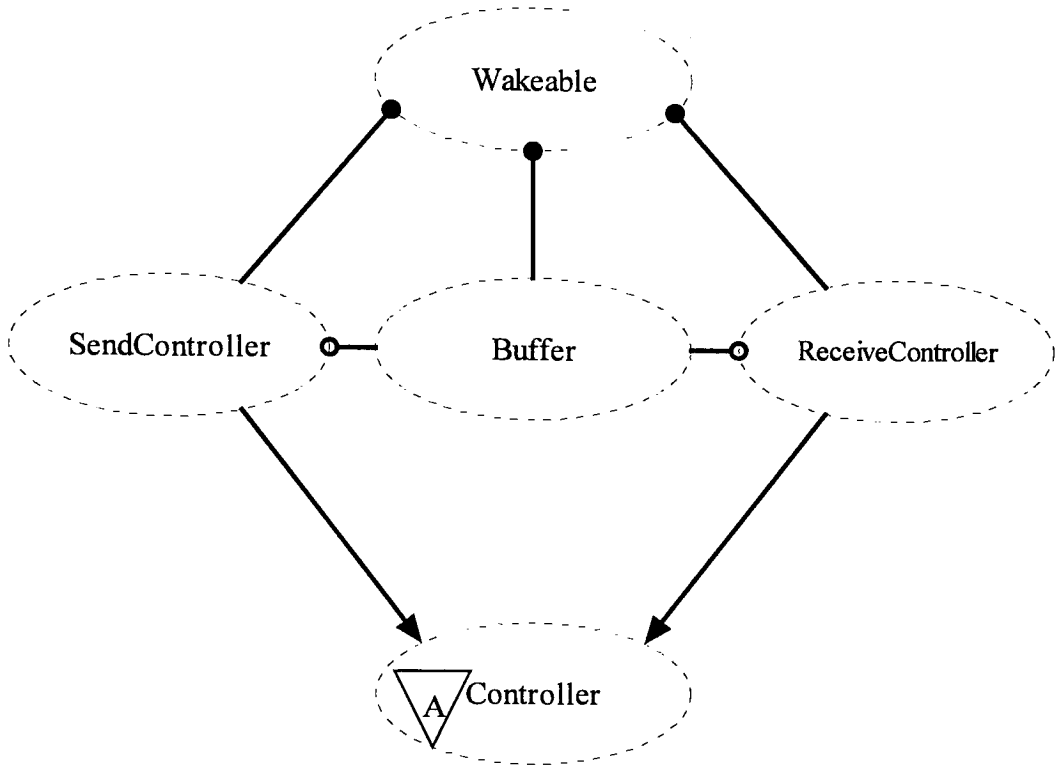

Figure 8 - Booch Diagram of Core Classes

4.3.1 Controller Class

The Controller class is an abstract class. An abstract class cannot be instantiated as an object. Instead, it provides functionality that is common to multiple classes. Those classes extend the abstract class. The abstract class becomes the superclass and all of its non-private methods and data become a part of the new subclass.

The abstract Controller class contains objects and methods that are used by all controllers. Both versions of Controller contain a buffer object , an IO object, and a Wakeable object (the Wakeable interface is discussed in section 4.3.4). The server

version contains a Feedback Object (see section 4.5.2). The client version has a RateCalculator object (section 4.4.2). Controller contains a method to set its internal buffer reference equal to one supplied as the method's parameter. There are also version specific methods used to set the internal RateCalculator or Feedback object equal to the one given as a parameter. In all cases, the previous object is lost when a new one is set.

A Controller implements the API supplied interface Runnable. In Controller, run is an abstract method and is not implemented. The implementation is deferred to the subclasses.

4.3.2 ReceiveController Class

ReceiveController is a subclass of Controller. It is responsible for receiving the data and placing it into the buffer. When a ReceiveController object is created, it is given an IO object. The ReceiveController begins to receive data from that IO when its run method is called. The run method invokes a method called TransmitData. That method loops for as long as new data is arriving. It looks for a header first. When the header signifies that no more data is on the way, the loop exits. Then, the ReceiveController calls its parent's Wakeup method and stops itself. If the header signals that more data will arrive, the object waits for the packet to arrive. When it does, a BufferObject is created. BufferObject is a class that contains a header, a byte array of size 1024 (the data packet), and an integer. That is the information received

for each packet. (The integer is only sent when the packet is not full. It contains the size of the non-full packet.) The BufferObject is placed into the buffer by the ReceiveController. The object then returns to the top of the loop and waits for the next header to arrive.

4.3.3 SendController Class

The SendController is the most important class in both the client buffer and the server buffer. The SendController's job is to send packets at some constant data rate. One packet must be sent every x milliseconds. To do this, the SendController has to take a BufferObject out of the buffer, remove its header and send it. Then, it must remove the packet and send that. Lastly, it sleeps until it is time to send the next packet.

The SendController starts when an object calls SendController's run method. The first thing it does is to perform the initial buffering for which the strategy calls. This is done by the method InitialBuffer. The method checks the first header received. If it signifies that the requested file was not found, the SendController sends that header on and terminates itself after calling its parent's Wakeup function. If it was a normal data header, the controller leaves the object in the buffer and sleeps for 500 milliseconds. After the sleep period ends, the method (really the thread running the object and its methods) awakens and checks the buffer's current size. If the current size is less than the requested initial fill size, the thread sleeps for another 500 milliseconds

and then checks again. It continues to sleep and check until the buffer is over the initial fill size (the default size is 200 packets). At that point, the loop terminates and the InitialBuffer method returns.

Once the initial buffering is complete, the object has to calculate the amount of time that it should sleep between sending packets. The time is based on the requested data arrival rate. The sleep time is calculated by the following lines of code:

```
float tempFloat = 1000 / _desiredRate;
_sleepTime = (int) tempFloat - 10;
```

The _desiredRate is the rate the application desires to receive data. The variable _sleepTime is the time in milliseconds the thread has to sleep between packet sends. The underscore symbolizes that a variable is part of the object. This differs from tempFloat, which is declared in the current method and its scope is only that method.

Java still has a number of bugs. One of them is to be in its sleep command. It is supposed to take an integer value and sleep for that number of milliseconds. Instead of doing that, the sleep will be for 10 to 15 milliseconds longer than the requested time. It will also be for a number of milliseconds that is evenly divisible by ten. Therefore, the sleep time has to be decreased by ten milliseconds to have the thread sleep for the correct amount of time.

The limitation of sleeping for increments of 10 milliseconds also limits the number of different rates at which data can be sent. If packets of 1024 bytes are used, the rates available will be those shown in Table 1.

Requested Sleep Time	Actual Sleep Time	Resultant Data Rate
0 ms	0 ms	~150 KB/s*
10 ms	20 ms	50.0 KB/s
20 ms	30 ms	33.3 KB/s
30 ms	40 ms	25.0 KB/s
40 ms	50 ms	20.0 KB/s
50 ms	60 ms	16.7 KB/s
60 ms	70 ms	14.28 KB/s
70 ms	80 ms	12.5 KB/s
80 ms	90 ms	11.11 KB/s
90 ms	100 ms	10.0 KB/s
190 ms	200 ms	5.00 KB/s
490 ms	500 ms	2..00 KB/s
990 ms	1000 ms	1.00 KB/s

* The data rate is limited by the overhead to approximately 150 kilobytes per second

Table 1 - Sleep Times with their Resultant Data Rates

Due to the truncation that occurs when converting a floating point variable to an integer, it is necessary to round up requests for fractional rates to the next highest integer. For instance, if the data rate desired is 12.5 KB/s the client must ask for 13 KB/s not 12. Asking for twelve would cause the SendController to send the data at ten kilobytes per second (the next lowest rate) instead.

Once the proper sleep time is calculated, the SendController enters a loop. The loop sleeps for the calculated sleep time and wakes up. It removes a BufferObject from the buffer, unpacks it, and sends the header and the packet to whatever program its IO has an open socket connection with. The loop continues until the "no more data" header is encountered. Then the loop ends and the SendController calls its parent's

Wakeup method. After that, the SendController calls its own thread's stop method and terminates.

In the client buffer's version of SendController the sleep time is only calculated once. However, in the server buffer's version the sleep time must be calculated every time a packet is sent. The calculation requires one more line:

```
float tempFloat = 1000 / _desiredRate;
_sleepTime = (int) tempFloat - 10;
_sleepTime += _modifier;
```

The value of _modifier is calculated immediately before this code fragment. The calculation is based on the value of the client's receive rate and its percentage full. Table 2 shows how those values affect the modifier.

Current Rate > Desired Rate	
Percent Full	Modifier
<= 10 %	- 100
<= 30 %	- 30
<= 50%	- 10
Current Rate < Data Rate	
Percent Full	Modifier
>= 75%	Send no data
>= 70 %	20
>= 60%	10
< 60 %	0

Table 2 - Determination of Modifier Values

The modifier values for the condition where the data rate is less than the desired rate are more severe than those for when the current rate is greater than the

desired rate. The modifiers were adjusted based on experimentation. It was determined that the danger of overflow was much less than the danger of underflow. The nature of a bursty network tends to limit the buffer from growing too quickly even when receiving data at too fast an average rate. Generally, a bursty network prevents the data rate from staying above the desired rate for more than a short amount of time.

4.3.4 Wakeable Interface

The Wakeable interface declares only one method, called Wakeup. A reference to an object that implements the Wakeable interface is provided to every Controller in its constructor. That Wakeable reference refers to the object that created the Controller. That parent object's thread sleeps after creating the controller. When the controller has completed its run, it calls the Wakeable object's Wakeup method and terminates. The Wakeable object can then continue its execution.

The Wakeable interface was created to avoid a coupling problem. In order for the send and receive controllers to inform their parent that they have completed execution they need a reference to that parent. To have one, they must be told the type of that reference. So, if they are told that their parent's class is ClientBuffer (see 4.4.1), then the Controller will be coupled with the ClientBuffer class. If someone wanted to use the controller on a completely unrelated project he could do so only if he created a class called ClientBuffer to instantiate the controller. This is an unreasonable restriction on code reuse. Instead, all the controller knows is that its parent implements

the Wakeable interface Since the interface can be implemented by any number of classes, any class that wishes to use a controller can do so. All it must do is to implement the Wakeable interface.

4.4 The Client Buffer

The client buffer program uses all of the core classes. In addition, three additional classes are used.

4.4.1 ClientBuffer Class

The first additional class, ClientBuffer, is the main class for the client buffer program. When a ClientBuffer object is created, a ServerSocket object is created as a part of that class. A ServerSocket is an object that is used to listen for a connection from a Socket. When a connection is made, a regular socket is returned by the ServerSocket. That socket becomes a part of an IO object,. When the Operate method of ClientBuffer is called, the ClientBuffer begins to run. It immediately begins to listen on the ServerSocket. Only when another process connects to the ServerSocket does the ClientBuffer execute further instructions.

First, the ClientBuffer creates some helper objects. A BufferMonitor, a Buffer, and a RateCalculator are created. The RateCalculator and BufferMonitor both have threads created and assigned to them but the threads are not started at that point. After those objects are created, the Client buffer receives the domain name and port number of the server from the program that connected to it. Using that information, an IO to

the network is opened. The IO asks the network to connect to the server at the requested domain and port. After the network connection is established, the ClientBuffer listens for the name of the file that is requested and for the required data rate. Both are sent over the network to the server. The requested data rate is also stored in the program.

Next, a SendController and a ReceiveController are created. They are given the IOs that contain the outside connections. The SendController receives the connection to the client. The ReceiveController is given the connection to the network. Neither controller is given a reference to the other. Therefore, any interaction between them is through shared objects, namely the buffer.

The ClientBuffer's last steps are to set thread priorities and to start the threads. The SendController is given the highest priority. Since that object must run at regular intervals, it is important that no other object be scheduled ahead of it when it wishes to run. The next highest priorities are given to the RateCalculator and to the BufferMonitor. They are set one priority level below the SendController. This is an arbitrarily chosen level. The priority for those two does not matter as long as they are set lower than the SendController and higher than the ReceiveController. The ReceiveController is set at the minimum priority level. Most of the time the ReceiveController's thread will be the only thread that is not blocked. All the others sleep for an amount of time, run for a few milliseconds, and then sleep again. The

ReceiveController, however, is always receiving data. It is the thread that is blocked the least often. Therefore it is give the lowest priority to ensue that it cannot monopolize the processor. That way all threads are given the opportunity to run. After thread priorities are set, the ClientBuffer starts all the threads and goes to sleep.

After the transmission is complete, the ClientBuffer resumes running. It stops all the other threads and destroys all the objects it created. This allows Java's garbage collector to free up the resources those objects used. After the cleanup is completed, the object begins to listen to a new connection. When one is received, the entire transmission process is executed again.

4.4.2 RateCalculator Class

The RateCalculator is used to calculate the average rate at which some event occurs. There is virtually no coupling between it and any other object. It can, therefore, be used to calculate the rate of any desired event. The calculator works as an active object. One thread is devoted exclusively to running the object. Other threads access public methods (which are never run by the calculator's thread) for all outside communication.

There are three public methods in the class. The first returns an integer that equals the most recently calculated value of the rate. The second method is called Mark. It takes no parameters and returns nothing. Each time the method is called, the elapsed time since the last method call is calculated. To avoid meaningless data, the

method does not calculate an elapsed time the first time it is called. The one hundred most recent elapsed times are stored in an array. Once one hundred have been stored, the oldest is removed and replaced with the newest. The third method can only be called once. It is called Finish. Once Finish is called the calculator will no longer calculate new rates and the last calculated value will not be available. Once finish is called the object should be destroyed.

The thread that runs the RateCalculator starts with the RateCalculator's run method. First run sleeps for one second. Then it calls the private method CalculateRates. This method sums the values of the elapsed times and divides them by the number of elapsed times that are in the array. The result is the average rate of the last one hundred events. This value is stored internally and is available to any object that calls the public get rate method.

In the client buffer program, a RateCalculator is used to determine the current average rate at which data is being received from the network. To do this, the ReceiveController calls the calculator's Mark method each time a header character is received from the network. The rate is sent to the server using the BufferMonitor class.

4.4.3 BufferMonitor Class

The BufferMonitor is another class whose instances are run as active objects. The BufferMonitor is the object that sends feedback to the server buffer. When it is

instantiated, it connects directly to the server buffer's Feedback object. That connection is used to feedback status messages about the client buffer.

The BufferMonitor sleeps for 500 milliseconds. Then it awakens and calculates the percentage of the buffer that is currently full. Next it gets the current rate from the RateCalculator, packs the two numbers into an array of two bytes, and sends that array on to the server preceded by a character header.

BufferMonitor has two public methods. The first sets the BufferMonitor's RateCalculator equal to the one given as a parameter. The second tells the monitor that the transmission is over and it should stop sending feedback. When this method is called, the BufferMonitor sends a header to the server buffer informing it that no more feedback will be arriving and sets an internal flag. Each time the thread running the BufferMonitor awakens, the done flag is checked. When it is set true, the BufferMonitor will terminate its thread by calling stop.

4.4.4 The Complete Client Buffer

The core classes combine with ClientBuffer, RateCalculator, and BufferMonitor to create the entire ClientBuffer daemon program. Once it is started it will not terminate during normal operation. The Booch class diagram for the whole program is in Figure 9.

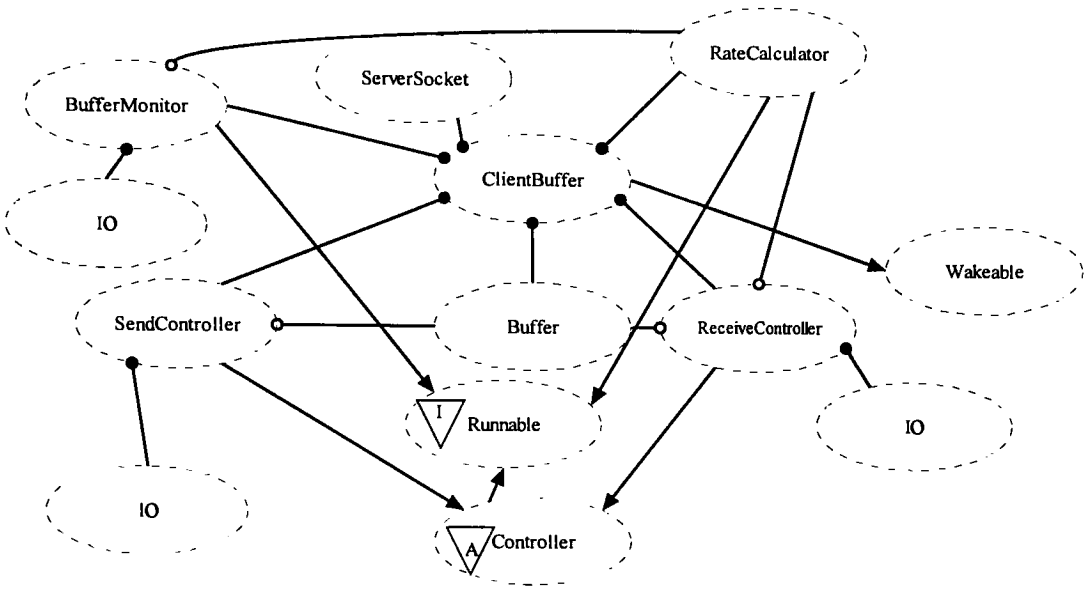

Figure 9 - Booch Diagram of the Client Buffer program

4.5 The Server Buffer

Like the client buffer, the server buffer uses the entire set of core classes. However, some modifications had to be made to them to compensate for the slight differences between the server buffer and the client buffer. Two changes had to be made to the SendController. There were also two changes made to the ReceiveController.

4.5.1 Changes to the Core Classes

4.5.1.1 SendController

Changes to the SendController were required for two reasons. The first was caused by the server buffer's need to change its send rate based on feedback from the

client. This required the addition of a reference to the Feedback class. This reference to the Feedback class was added to the abstract Controller class. Also, added to that class was a SetFeedback method similar to its preexisting SetBuffer. The change made directly to SendController was the addition of the CheckFeedback method. This method checks the most recent feedback values. Based on the feedback values, the sleep time modifier is calculated (see Table 2).

The second SendController change is the addition of the ServerFeedback class and its thread. This class is a requirement of the network simulator and is used to workaround a feature of Java's data streams that interfered with the network simulator's operation. The problem and solution will be discussed in chapter 5.1.2.2.

4.5.1.2 Receive Controller

The two changes to the ReceiveController were relatively minor. The first is the elimination of the call to a RateCalculator's Mark method whenever a header arrives. There is no need for a RateCalculator in the server buffer since there is no need to determine the rate at which data arrives.

The second change is the addition of a maximum buffer size. Testing showed that while an infinite size buffer worked in theory, there was only enough memory for about 7000 packets to be buffered. Therefore, a limit of 5000 packets was added. If the buffer climbs over 5000 packets in size, the Server is told to temporarily stop sending packets. When the buffer size drops below 3000, the server is allowed to resume

sending. This lower restart threshold avoids the scenario where a "do not send" message is followed by a "send" message which is quickly followed by a "do not send" message when one packet arrives. That situation would cause the buffer to oscillate around 5000, while wasting resources by sending large numbers of messages back to the server.

4.5.2 ServerBuffer Class

ServerBuffer serves the same purposes for the server buffer as ClientBuffer did for the client buffer. It is the main class and the only class that should be directly instantiated. It creates all the other objects when necessary.

The ServerBuffer begins by instantiating a Feedback object. A thread is created for the object and the thread is run immediately. It is necessary that the Feedback thread be listening for feedback connections before the server buffer is accepting regular file request connections. This avoids the scenario where the client buffer opens both connections before the Feedback object can be created. If that were to happen the ClientBuffer would throw an error and terminate.

After the Feedback is established, the ServerBuffer follows a pattern like the ClientBuffer. First it blocks, listening for a connection. This time, however, the connection is coming from a ClientBuffer (through a network). When a connection is made, the ServerBuffer continues executing. The first step is to get the filename and desired data rate from the network. With that information, the ServerBuffer connects to

the server program running on the same machine. The server is given the name of the desired file. After the connection is established, a SendController and a ReceiveController are created. Their priorities are set to the same levels as in the ClientBuffer and their threads are started. Then, again, just like ClientBuffer, ServerBuffer sleeps until it receives wakeups from both controllers.

After both wakeups are received, ServerBuffer eliminates the Controller objects and closes the connections. It loops around and begins to listen for a new connection.

4.5.3 Feedback Class

The Feedback class maintains three internal variables. They are _status, _rate, and _percent. Each holds the corresponding value most recently received from the client buffer's BufferMonitor. All three have private set methods used to set their values. There are also public get methods that allow another object to obtain their values. All six methods are synchronized. Synchronized methods require that an exclusive lock be obtained on the object before the method can be run. When the method is complete the lock is released. The locks prevent scenarios where the get and set methods for a variable are accessed at the same time. Although there does not appear to be a way that this could cause deadlock, proper design suggests that using synchronized methods is still a good precaution.

The Feedback object's run method is short. First, it listens for a client buffer to open a feedback connection. When run is called, the method enters a loop where it listens for new values for the variables. It waits to receive values for status, percentage, and rate – the percentage and rate arrive in a byte array – and then it calls the appropriate set methods. After receiving the two pieces, header and array, the method waits for a new header. When it receives the "done" header, the loop is terminated and the receive IO object is destroyed. Destroying the IO also closes the connection. After that, it loops back and listens for a new connection.

4.5.4 Complete Server Buffer

The server buffer operates as a daemon. Therefore, like the ClientBuffer, it will not terminate unless an error occurs. The complete Booch class diagram for the program is in Figure 10.

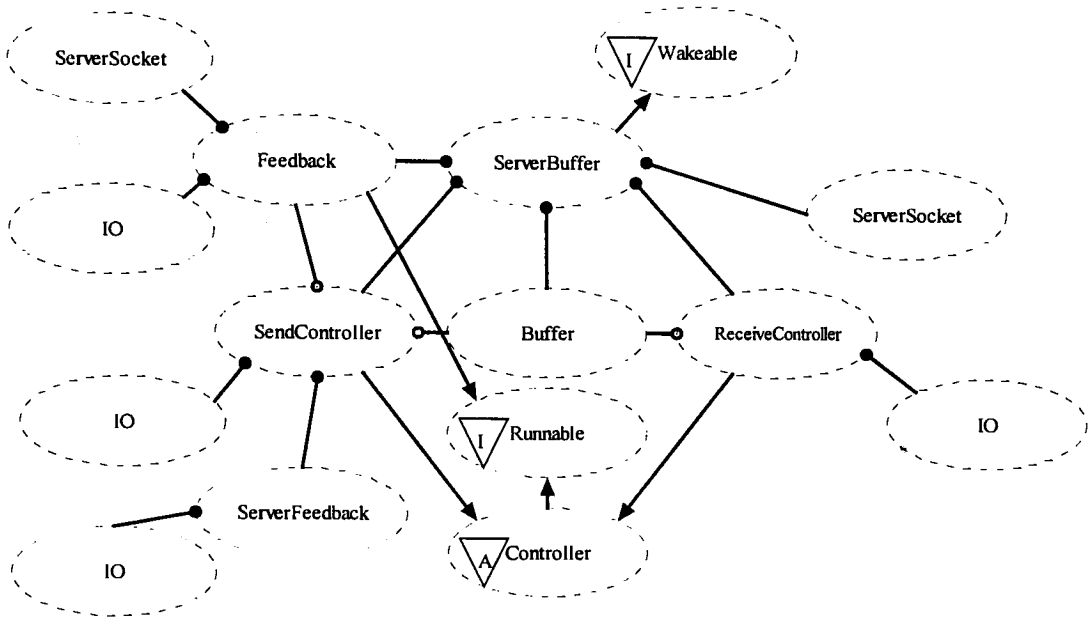

Figure 10 - Booch Diagram for the Server Buffer Program

5.0 Results and Analysis

5.1 The Network Simulator

When the strategy was initially proposed, some consideration was given to how the Java implementation was going to be tested. Two possibilities were proposed. The first idea was to run the programs over the Internet. To do so would have required a remote account on some system. The server buffer could have been run at the remote site. The client buffer would be run at RIT.

The problem with this approach is the uncontrollable nature of the Internet. The Internet is a dynamic system with changing traffic patterns and loads. Although these are some of the reasons that the strategy was initially proposed, such characteristics make test results difficult to reproduce. It would also be difficult to determine the conditions of the Internet during a test. That would make it impossible to determine which aspect of the network caused the programs to run as they did. Therefore, it was decided that using the Internet for initial testing was not the best solution.

The second approach called for the development of a network simulator whose conditions would be controllable. The simulator could be run on the same local network as the buffer programs. The simulator would connect the client buffer to the

server buffer. Then, it would turn the server buffer's constant stream into a bursty traffic pattern for reception by the client buffer. This is the approach that was chosen.

To implement this simulator required two programs. The first was a server program that would read a file from disk and send it over the network. The second was the network simulator program itself. In addition, a simple client application was written that could play .au (Sun's audio format) files as they are received from over the network.

5.1.1 The Server

The Server program is fairly elementary. When created, the server object creates a ServerSocket and listens to it. When a connection is opened, the server takes the corresponding IO and waits for a string to be received. It expects to receive the name of the file that the client wants. When that information is received, the Server tries to open the file. If the file cannot be opened, the "no file found" header is sent to the client and the Server closes the connection. Then, it begins to listen for a new connection.

If the file is successfully opened, the Server's next step is to create a ServerFeedback object and a thread to run that object. A ServerFeedback object maintains one status variable. It also provides a public get status method used by the server to check the status. The status variable is set based on the last feedback sent by the program that contacted the server (either a Network or a ServerBuffer). The status

variable has two possible values, "stop" and "go." When "stop" is received, the server stops sending packets. When "go" is received, it resumes sending them.

After starting the ServerFeedback thread, the server begins to send packets. Packets are sent as arrays of 1024 bytes and are preceded by a one character header. Before each packet is sent, the server checks the current value of status. If it is "stop" the server will not send another packet until the status changes to "go." Each packet is read from the file input stream immediately before sending it.

When the entire file has been sent, the server sends one last header, the "no more data" header. Once it is sent, the server closes the connection, Then it stops the ServerFeedback's thread and eliminates the object. After that is done, it begins to listen for a new connection

5.1.2 The Network

5.1.2.1 Network Class

Network is a class that simulates a perfect network. That means that the entire bandwidth is available to the transmission. There are no bursts. Although it is possible to use Network directly, it is not really intended to be used that way. Instead, another class should extend Network and overwrite its TransmitData method with a version that implements the traffic pattern that is to be simulated.

When a Network is instantiated, it listens to a ServerSocket. When a connection is made by a client, the Network waits for the server's host name and port.

After it receives both items, an IO is established with a connection to the server. Then, the Network receives the name of the file to be sent by the server. The name is passed on to the server. Lastly, the network waits for the required data rate and sends that to the server.

Once all the information necessary to get the server running is received and sent, Network calls its TransmitData method. This method contains a loop that blocks and waits for a header. When one is received it is sent to the client. Next, the method waits for a packet. When the packet is received, it is also sent to the client. This continues until a "no more data" header arrives. That header is sent on to the client and the TransmitData method ends.

The Network, and therefore the network simulator, does not simulate network latency. The latency of a network is the amount of time it takes data to travel over the network. Even under the absolute best circumstances, information cannot travel faster than the speed of light. Therefore, there must be some delay between when the server sends a packet and the client receives it. The only delay in the simulator is that which is caused by the overhead in Network.

The only effect of not simulating network latency is the data stream arrives without a delay. The average latency should be the same for every packet. Any changes in latency for the individual packets will simply add to the bursty pattern.

Therefore, the lack of a wait for the first packet to arrive at the client application will be the only noticeable effect of the lack of network latency.

The network simulator also does not handle the client's feedback. Instead the BufferMonitor class directly contacts the Feedback class over the real network. This simplification made both the network simulator and the feedback easier to implement.

5.1.2.2 TimeNetwork and TimeNetworkWrap Classes

The TimeNetwork and TimeNetworkWrap classes are used to simulate a bursty network. They do so by only transmitting data for a certain percentage of each second. During the rest of that second, the network is unavailable to the transmission and no packets are sent.

The TimeNetwork class extends Network and overwrites its TransmitData method. It also adds two methods and one internal variable. The variable is a flag that keeps track of whether or not the network's bandwidth is available to the transmission. It is set using the SetAvailability method. A GetAvailability method is provided as well.

TimeNetwork's version of TransmitData is the same as Network's except for two changes. Before the method tries to receive the header, it checks the availability flag. If it is true, it proceeds to receive the header. If it is false, the method will keep checking the flag until it is true. The method will start sending data again when the flag becomes true.

The second change is the opening of a connection to the server's (or the server buffer's) ServerFeedback object. When the availability flag is false, the ServerFeedback object is sent a message telling the server to stop sending data. When the network becomes available again, feedback is sent telling the server to resume.

The addition of the ServerFeedback object to the server and server buffer programs was made necessary by the high level nature of Java. Java attempts to make socket communications as smooth as possible. Therefore, when TimeNetwork sleeps and is not receiving data, Java allows the server to continue to send data to the TimeNetwork. Since the data is not being received, it is buffered in an under the hood buffer. When the TimeNetwork begins to transmit data again, there are a huge number of packets buffered. This results in the network sending data to the client faster than it is receiving them from the server buffer. The ultimate effect of this chain of events is to make the changing in the server buffer's send rate ineffective. Because of all the buffered data, changing the send rate has no effect on the rate the client buffer receives data. The ServerFeedback object is used to eliminate this problem.

The TimeNetworkWrap class serves three purposes. The first is to serve as the program that is run at the command line. It takes as an argument a percentage in integer format. It is the percentage of a second during which the network should be available. That percentage is not passed directly to the TimeNetwork. The second job

TimeNetworkWrap has is to create a TimeNetwork object and a thread for the TimeNetwork to be run. That thread is then started.

The last responsibility of the TimeNetworkWrap is done in an infinite loop. In that loop, the thread sleeps for the number of milliseconds that the network should be available. Then it wakes up and calls the TimeNetwork object's SetAvailability method and sets it to be false. Next, the thread sleeps for the remainder of a second. Then it wakes up and sets the network availability to true. The loop then returns to the top and sleeps for the number of milliseconds the network is available.

The two classes work together to make the bursty network simulator. The Booch class diagram for the entire simulator is in Figure 11.

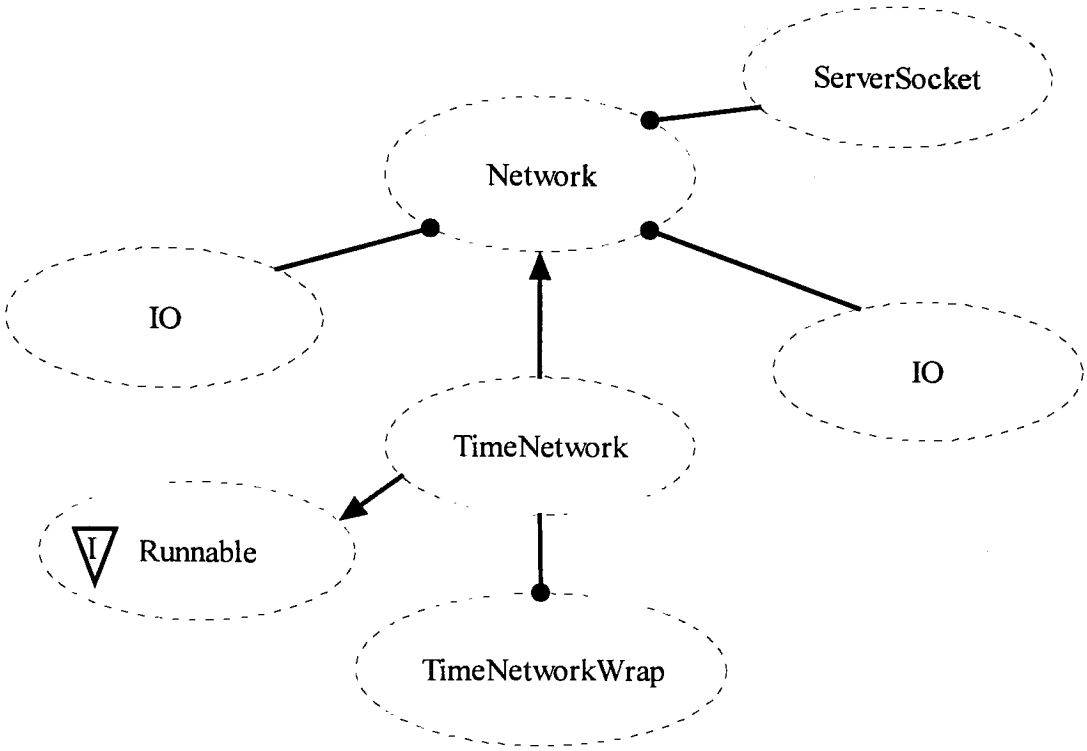

Figure 11 - Complete Bursty Network Simulator

5.1.3 The Sound Player

The sound player is used as a client program. It takes, as command line arguments, the name of the file to be retrieved and the data rate at which it should be received. With that information, it contacts the client buffer and requests the transfer. When packets start to arrive it sends them to the audio_daemon that does the actual sound playing. The program also stores an array of elapsed times between packet receptions. This array is passed to the Statistic class' mean and standard deviation methods. The results of those calls are printed at the end of the program along with the total elapsed time.

A third command line argument can be optionally given to the sound player. If the option "-nobuff" is placed after the file name and the data rate, the player will contact the network directly and ignore the server and client buffers. All other parts of the program remain the same. This additional mode of operation allows comparisons of transmissions that used the strategy with transmissions that did not.

5.2 Results

To test the implementation, a number of tests were run using the network simulator. Tests were run for client buffers of size 1000 and 250. These values were chosen to test a large buffer as well as for one that only holds a little more data that is initially buffered. Initially, one has more room for further buffering while the other is nearly full. Tests without the buffers were also run for comparison. All tests were run with the server buffer and server program on the same computer. The network simulator was on a second. The client buffer and client player were located on two separate but physically adjacent machines. This kept the programs off the same processor. This way they did not interfere with each other.

5.2.1 Testing the 1000 Packet Client Buffer

The sound file used for the tests is 663,556 bytes long and is sampled at 8 kHz. That means that the audio player needs to receive data at 8 kilobytes per second. The slow rate causes a minor problem. Eight KB/s is too slow a rate for a meaningful test.

It will not tax the network simulator sufficiently to prove that the client buffer is working. Unfortunately, the Java language only supports 8 kHz files at this time.

To solve that problem, the server was modified to send each data packet three times. This required that the sound player be modified to only play every third packet received. Sending 2 extra packets for every real one increases the necessary data rate to 24 KB/s. Looking back at Table 1, 25 KB/s is the closest rate that can be requested. Therefore, 25 KB/s was the rate requested for all tests.

Padding the file makes it seem to the client that a 1.944 megabyte file is being sent. That is 1944 whole 1024 byte packets plus a fraction of another. After all packets have been received, a simple statistical analysis of the elapsed times between packet receptions is calculated. The first test used a client buffer size of 1000. The tests were run for all cases where the network percentage available was evenly divisible by ten. Five percent network availability was also to provide another test when the network throughput is very low. Each test was run five times. The individual results of each test were averaged. The averages were used as the final results and are in Table 3.

Percent of Network Available	Mean Elapsed Time (ms)	Standard Deviation	Total Time (Sec.)	Mean Data Rate (KB/s)
100	41.19	8.96	80.03	24.87
90	40.96	5.80	79.58	25.01
80	41.25	7.98	80.14	24.84
70	41.29	7.61	80.22	24.81
60	41.33	7.99	80.32	24.78
50	41.44	8.45	80.52	24.72
40	41.50	9.87	80.64	24.68
30	41.20	6.98	80.06	24.86
20	42.01	9.92	81.67	24.37
10	49.32	73.90	95.45	20.86
5	101.61	214.04	197.43	10.08

Table 3 - Results with 1000 Packet Client Buffer

If data is received at 25 KB/sec (25 packets per second), then the nominal correct elapsed time is 40 ms. For a 1.944 megabyte file, the total time should be 79.6 seconds. Looking at the Table 3, the results are slightly worse than the correct rates. The mean elapsed times are about a millisecond worse and the total time is usually just under a second more than is expected. This error can be explained by the inexactness of the method used to determine elapsed times.

Elapsed times were measured using Java's Date class. When a Date object is created it is stamped with the current time. The time is stored as the number of milliseconds since midnight GMT, January 1, 1970 (Flanagan, 1996). Every time a packet arrives, a new Date is created. The previous Date's millisecond value is

subtracted from the new value to arrive at an elapsed time in milliseconds. This results in a fairly precise measurement. However, the new date is created each time the code reaches a certain command in the loop. The majority of that loop is spent waiting for a packet, which is the time that should be measured. The rest, however, is spent on the overhead required to receive the packet and send it to the player. The time spent on that overhead is enough to add extra time to the measurement. The total time is the sum of all the elapsed times. The overhead error is propagated into that measurement.

Occasionally a packet is late arriving at the player after traveling from the buffer. The frequency that this occurs during one test can be compared with other tests by comparing the standard deviation of the elapsed time between packet receptions. The higher the deviation, the more the data is spread out. During playback, late packets are heard as short, silent periods that sound like a record skipping over a scratch. These skips can even happen when the client buffer has plenty of data available to be sent. This means that the buffer is still sending at the constant rate but the packets are not being received on time.

Observations during testing suggest that the cause of this problem is task switching on the processor. Unless the operating system is single-threaded, the sound player will never have full use of the processor. It has to share that resource with other processes (including the operating system). If the load on the machine playing the audio file is high, the sound player may not have the processor when a packet arrives.

If that happens then the packet will not be played on time and there will be a skip. To limit processor interruptions to the extent possible, the tests were run with no competing user processes on the two client machines. All results indicate that the lack of other user processes helped decrease the frequency of the skips.

Although the elapsed times are slightly high, there was no effect on the quality of the playback. Other than the skips mentioned above, the quality of the playback was high when the bandwidth was high. The quality remained good until the bandwidth available was decreased to 10 percent of the maximum. At that point, the server buffer could not increase the send rate further to make up for the worsening network. The server program feeds the server buffer at approximately 150 kilobytes per second. Twenty percent of 150 is thirty. If the server buffer sends at 150 KB/s but the network only transmits data for 20 percent of the time, then the client buffer will receive 30 kilobytes every second (in short bursts). That is an acceptable rate. However, ten percent of 150 is only 15 KB/s. That is not enough packets per second to supply the application. The network had dropped below the point where the buffers could help. If the network continues to degrade, the results will get worse. This is observed at the five percent level.

If the available bandwidth of the network is ten percent, then the expected data rate is 15 kilobytes per second. From Table 3, the average data rate ten percent was 20.86 KB/s. There are two possible explanations for this. One is the initial buffering.

No matter how bad the network conditions, the first 200K will be buffered and sent to the client application at the correct rate. That provides a temporary high quality of service before the bottom falls out when the buffer empties. The initial period may be enough to increase the average data rate.

The second possibility is related to the network simulator. Due to Java's thread scheduling and sleep command, the percentage of the network available is often slightly different from the requested percentage. When the buffering strategy is working, this effect is masked by the client buffer's rate stabilization. When the client buffer can no longer supply the requested rate, the effect cannot be masked and will appear in the results.

5.2.2 Transfers Without the Strategy

To determine how effective the buffering strategy was in improving the quality of service, the same tests were run for the case where the client and server buffers were not used. The sound player was connected directly to the network simulator. That program connected directly to the server. The results are in Table 4.

Percent of Network Available	Mean Elapsed Time (ms)	Standard Deviation	Total Time (Sec.)	Mean Data Rate (KB/s)
100	41.85	10.69	81.33	24.48
90	45.81	19.60	89.02	22.36
80	57.42	48.53	111.57	17.84
70	65.97	61.58	128.18	15.53
60	81.74	93.16	158.82	12.53
50	130.10	161.28	253.61	7.85
40	242.90	220.42	471.96	4.22
30	297.42	233.68	577.89	3.44
20	330.60	263.51	642.47	3.10
10	335.26	227.22	651.40	3.05
5	331.84	232.15	645.03	3.09

Table 4 - Results Without Buffering

With 100 percent of the network available the results are fairly good. But with the network at 90 percent, the results are worse than when the network is twenty percent with no buffering. With further reductions in throughput, the mean data rate plummets and the total transmission time increases substantially. A file that should take less than 80 seconds to play takes over ten minutes when 20 percent of the network bandwidth is available. As high as fifty percent, the time is twice correct amount. Since these results are the ones the buffers work to correct, it becomes apparent that the buffering strategy worked quite well.

The quality of the play back was noticeably bad for almost all the tests. At 100 percent it sounds fine, but by 90 there are short but regular skips when no data is

received. This level of quality is tolerable but not great. By 50 percent, the skips are closer to pauses. Individual words are still intelligible but the overall quality is not very good. Indeed, it can get rather annoying. At twenty or thirty percent, the pauses are dominant. It can take three or more seconds for one word to be played. Listening to a song at this level is quite intolerable. It is worse than any of the buffered levels..

5.2.3 A Smaller Client Buffer

The results were quite good when using a 1000 packet buffer. However, such a buffer requires that over one megabyte of memory be allocated to the buffers. This is not a large amount of memory today. However, it is always best for a program to use the minimum amount of resources possible. To test whether the strategy will work using a smaller buffer the same tests were run using client buffer that only held 250 packets. The results are in Table 5.

Percent of Network Available	Mean Elapsed Time (ms)	Standard Deviation	Total Time (Sec.)	Mean Data Rate (KB/s)
100	40.84	5.54	79.30	25.10
90	40.67	5.30	79.02	25.19
80	40.88	6.17	79.44	25.06
70	41.13	6.34	79.91	24.91
60	41.27	7.40	80.20	24.82
50	41.72	20.15	81.46	24.44
40	41.82	20.42	81.25	24.50
30	41.48	15.98	80.33	25.78
20	43.88	50.16	85.25	23.35
10	54.49	90.91	105.87	18.80
5	96.71	217.36	187.91	10.59

Table 5 - Results With a Smaller Buffer

The values for higher network percentages are slightly better than in the first test. However, this does not appear to be meaningful. The improvement is not significant enough to suggest anything more than a slightly lower system load. Comparable results were observed with a 1000 packet observer during functionality testing.

At 50 percent and below, the standard deviation increases dramatically. This appears indicative of the danger in using a small buffer. The server buffer's algorithm works to maintain the client buffer at 50 percent full. This appears to be the optimal percentage for buffer operation (Cowan and Cen, 1995). As the network conditions

deteriorate, the more difficult it is for the server buffer to maintain the client buffer at fifty percent full.

Fifty percent of one thousand is five hundred. Fifty percent of 250 is only 125. The point where the server buffer begins to scale back the transmission to avoid overshooting the 50 percent target is also a smaller number (300 versus 75). Even though the network's percentage available is constant throughout a transmission, the amount of data that makes it through to the client during a burst is somewhat variable. The variation is caused by inexact nature of thread scheduling for both the network simulator and the server buffer. This variation can lead to conditions where the effective data rate received by the client buffer drops. The drop in rate causes an emptying of the client buffer until the server buffer notices the change and refills the buffer. If there were three hundred to five hundred packets in the client buffer, the drop will have little effect. However, if there were only one hundred packets in the buffer, a long enough drop could empty the buffer. This would cause a major delay in the client application's reception of packets.

During the tests, this kind of drop would happen in one to three of the five runs when the network throughput was below 50 percent. Since it would only happen once or twice per test run, the mean times will not be very heavily affected. One or two delayed packets out of 2000 will not affect the mean significantly. The standard deviation, however, will be affected. By the time throughput is twenty percent, a

collapse occurs during enough runs that the standard deviation has increased significantly and the mean elapsed time is starting to be affected. At ten percent, the buffers are no longer able to compensate for the network and all the statistics are bad.

Because of this problem for throughputs below 50 percent, using a larger buffer is a good idea. The extra memory used is worth the extra cost due to the improved reliability of the client buffer. This is especially true since a real network's throughput can vary during a transfer. If it drops temporarily below twenty percent, the extra hundred or more packets in the buffer will enable the client buffer to compensate for a longer time before underflowing.

6.0 Conclusion and Future Work

6.1 Conclusion

The goal of this thesis was to present the buffering strategy and to show that the strategy was capable of improving the quality of service for networked multimedia applications (and any other application) that can be hurt by bursty network traffic. To do so, a Java language based implementation was performed. The results seem to indicate that the strategy will work to provide a constant rate long after the network has ceased to provide decent quality.

The strategy, as outlined in chapter three, suggests that the performance of a networked multimedia application should be improved when using the strategy. The average data rate received should be constant and stable. The Java implementation presented here accomplished those goals. When 30 percent of the simulated network was available, the bufferless playback took 297 seconds and was nearly unintelligible. The buffered playback took 80.06 (less than half second longer than the best case) and had only a few skips, which were caused by the processor's context switches.

The implementation and results prove that the strategy is valid. However the results do not necessarily prove or disprove that this implementation would work with a real network. To do so would require that the network simulator be replaced with direct socket connections between server and client. Testing under those conditions

would prove whether the Java implementation will work under real network conditions.

Another thing to be considered is the choice of Java to implement the strategy. The object oriented nature of Java made designing and implementing the strategy easier than a non-object oriented approach would have been. There are, however, other object oriented languages (C++, smalltalk) available.

Java's problems are its high level nature, its interpreted nature, and its buggy state. The high level forced a few additions that complicated the program (ServerFeedback class). Because it is high level, sleep does not work for times less than a millisecond. Therefore megabyte per second transmissions are not possible. A lower level language may be able to supply sleep times lower than a millisecond. It is possible, however, that the operating system could limit most or all languages' ability to sleep for times much less than a millisecond. If this is true, then a new method for determining when to send a packet would need to be developed.

The interpreted nature of Java slows the execution of some Java programs. The biggest example of this is Java's inability to quickly process mathematically intensive algorithms without using native code. Because of this, video decoding is generally considered to be too much for Java to handle.

The bugs in the language cause two problems. They can be a significant inconvenience, often forcing programmers to use ugly workarounds. For example, the

problems with the sleep command greatly decreased the number of data rates at which the application could be run.

Despite the problems, using Java to implement the strategy accomplished what was intended. The implementation demonstrated that the buffering strategy is capable of greatly improving the performance of a networked multimedia application. The API provided time saving socket classes that saved more time than the bugs cost. However, it is probable that a different, lower level language should be used to develop an implementation for general use.

6.2 Future Work

Although the strategy has been proven, there are still steps that need to be taken to make it completely viable for general full time use. Further tests of the Java implementation are needed. In addition, a general purpose implementation would probably require some redesign.

The first thing to be done is to test the current implementation over the Internet. This would require modifying the buffer programs to have the client buffer to connect directly to the server buffer. A remote account would be needed and any permissions for remote contact would have to be set up.

Another possibility would be to re-implement the strategy in a different language. The design could remain the same but the actual code would have to be completely changed. It would be best if a somewhat lower language were used. C++ is

a possibility as it has more direct access to system resources. Unlike Java, it does not have an interpreter and some system threads working underneath it, trying to improve stream flows. Because there is no interpreter, the buffer programs would run faster. This would allow the buffers to supply faster data rates to the client. However, the implementation would take longer due to the need to spend time implementing classes that would be equivalent to classes in the Java API.

The final step would be to use the information learned in previous implementations to develop two complete programs, one server buffer and one client buffer. The idea is to develop programs that are suitable for release to the public. To meet that criteria, they would need to support a much larger range of data rates and be able to be run as daemons. They would need to have an interface to applications and to the network that is transparent to those programs. Applications that were not designed with the strategy in mind should be able to use it, even without knowing that it is there.

Such programs would be the final step toward making the strategy useful for users of all networked applications. If the programs were developed they should greatly increase the usefulness of that class of applications. With an interface to the network that can assure a stable constant rate, the user can be assured a high quality of service without having to use specially designed client and server programs that only work with each other.

Bibliography

Booch, Grady, Object-Oriented Analysis and Design with Applications, New York: The Benjamin/Cummings Publishing Company, Inc., 1994.

Cen, Shanwei, Pu, Calton, et. al., "A distributed Real-Time MPEG Video Audio Player," Appeared at The Fifth International Workshop on Network and Operating Systems Support of Digital Audio & Video (NOSSDAV '95), April 18-21, 1995, Durham, NH. Available at ftp://cse.ogi.edu.pub/dsrg/synthetix/nossdav.ps.gz.

Cowan, Crispin, Cen, Shanwei, Walpole, Jonathan, and Pu, Calton, "Adaptive Methods for Distributed Video Presentation," in ACM Computing Surveys, 27 4 (December 1995): 580-583.

Flanagan, David, Java in a Nutshell, Sebastpol, California: O'Reilly & Associates, Inc. 1996.

Furht, Borko, "Multimedia Systems: An Overview," In IEEE Multimedia, 1 (Spring 1994): 47-59.

Little, Thomas D. C. and Venkatesh, Dinesh, "Prospects for Interactive Video-on-Demand," In IEEE Multimedia, 3 (Fall 1994): 14-24.

Minoli, Daniel and Keinath, Robert. Distributed Multimedia through Broadband Communications, Boston: Artech House, 1994.

Patterson, David A. and Hennessy, John L., Computer Architecture, A Quantitative Approach 2nd ed., San Francisco: Morgan Kaufman Publishers, inc., 1996.

Tanenbaum, Andrew S., Computer Networks 2nd ed., Englewood Cliffs, NJ: Prentice Hall, 1989.